Partitioned Garbage Collection of a Large Stable Heap (Extended Abstract)

Barbara Liskov Umesh Maheshwari Tony Ng

MIT Laboratory for Computer Science, Cambridge, MA 02139

Abstract

This paper describes a new garbage collection scheme for large persistent object stores. The scheme makes efficient use of the disk and main memory and does not delay applications. The heap is divided into partitions that are collected independently using stable information about inter-partition references. We use novel techniques to maintain this information using in-memory data structures and a log to avoid disk accesses. The result is a scheme that truly preserves the localized and scalable nature of partitioned collection.

1 Introduction

We present a new technique to collect garbage in large persistent object stores. Such storage, also known as a stable heap, is found in many object databases, persistent programming language environments, and distributed shared memory systems. In these systems, the heap resides on the disk because it is much larger than the main memory and must be recoverable after a crash. Applications access the objects through a memory cache and log updates for crash recovery.

Schemes that trace the entire heap (e.g., [2, 8, 18]) do not scale to very large heaps because the non-local nature of garbage collection causes random disk accesses. Therefore, some systems partition the heap into independently collectible areas [4, 7, 21, 1, 15, 5]. This is also the approach taken in many distributed system, e.g. [10, 9, 13, 19]. Generational collectors are a variant of partitioned collection that use the ages of objects to optimize the collection of younger, smaller partitions [11]; however, the age-based heuristics are not applicable to persistent stores [3].

A problem with partitioned collection is the efficient maintenance of information about inter-partition references, which is needed to trace partitions independently. For a large heap with many partitions, and also for fast crash recovery, the information must reside on disk, and care is needed in reading and updating it without degrading performance. Increasing the partition size helps reduce inter-partition references, but tracing large partitions slows both the garbage collector and the applications due to increased contention for the cache and disk [1].

We present a new partitioned scheme that uses a log and in-memory data structures to provide the following benefits:

1. Disk accesses for reading and updating information about inter-partition references are deferred and batched.
2. Reading objects from the disk and evicting them from the cache do not require processing garbage collection information or reading it from disk.
3. The in-memory data structures are compact yet available in an efficient form.
4. The scheme is fault-tolerant; collection information is recovered quickly after a crash.

The overall effect is that the collector makes infrequent and localized accesses to the disk. In particular, to collect a partition, only that partition and the information about inter-partition references concerning its objects, need be read/written to disk. The scheme has been partially implemented within Thor [12, 17]. We have also extended it with a global incremental marking scheme that allows collection of inter-partition cyclic garbage [14].

One other scheme, PMOS [15], makes use of a log to defer and batch processing of information about inter-partition references. We compare the two approaches in Section 4.

The remainder of the paper is organized as follows. Section 2 describes the system model. Section 3 describes partitioned collection. Related work is discussed in Section 4. We close in Section 5 with a discussion of what we have accomplished.

2 The Model

We assume a system in which the heap resides on disk, while applications access objects in the heap through a main-memory cache. Modifications to objects are recorded in a write-ahead log that is forced to stable storage as needed; the log allows the heap to be recovered in a consistent state after a crash. We assume that the head of the log is cached in primary memory even if it has been forced to disk; it

Contact: umesh@lcs.mit.edu.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136.

1063-5351/96 $5.00 © 1996 IEEE
Proceedings of IWOOOS '96

is truncated when it grows too big. Similarly, when the stable log gets too big, it is truncated after ensuring that the modifications have been installed into the stable heap.

Objects are clustered in *segments* on disk. Like a page, a segment is stored contiguously on disk and is the unit of disk access. A segment provides opaque references for its objects, so that objects can be compacted within their segment without having to update references to them stored in other objects (as in [1]). References need not be completely opaque; for example, in Thor, a reference contains the segment number of the referenced object so that objects can be located efficiently without a global object table.

Objects in the heap may contain references to other objects. Applications navigate by starting at some *persistent root* object and may read or modify the objects they reach. They may also store references to objects in local variables. There could be a single application thread accessing the cache directly, as in persistent programming language environments, or there could be multiple application threads, as in a client-server system, where applications access the server cache through a higher level interface and may have caches of their own.

The job of the collector is to reclaim storage allocated to objects that are useless because they are not reachable from the persistent root or any application variables.

3 The Scheme

This section describes our scheme. The first few sections largely ignore issues of fault tolerance; fault tolerance is discussed in Section 3.5.

The heap is divided into partitions, each of which can be collected independently. A partition is chosen to be an efficient unit of tracing. There is a tradeoff here: Small partitions mean more inter-partition references and also more inter-partition cyclic garbage. Big partitions mean more cache space used by the collector and possibly disk accesses during tracing. Our scheme provides mechanisms to handle inter-partition references so that partitions that fit in a fraction (say, a tenth) of the primary memory may be used efficiently.

Partitions contain several segments, possibly non-adjacent. Decoupling partitions from segments has important advantages. First, a partition can be much bigger than a segment. For example, a partition could be several megabytes while segments could be tens of kilobytes. Segment size is chosen to allow efficient fetching and caching; to service an application cache miss, only the required segment is read in. Second, it is possible to configure a partition by selecting a group of segments so as to minimize inter-partition references—without reclustering objects on disk. Furthermore, partitions can vary in size, while segments provide a fixed-size unit of disk access. For example, a partition can represent the set of objects used by some application, and the size of the partition can be chosen to match the set.

We assume that, given a reference to an object, its partition can be computed efficiently. We do this by having the reference contain its segment number and keeping a cached table that maps segments to partitions and vice versa.

3.1 Inlists and Outlists

To collect a partition independently of the rest, the collector remembers the objects in the partition that are referenced from other partitions, and uses them as roots. We call this information the *inlist* of the partition. An inlist contains a list of references with associated reference counts. The reference count is the number of other partitions that contain one or more copies of the reference. When a reference count drops to zero, the entry is removed from the inlist.

To efficiently update inlists as inter-partition references are created and deleted, we also maintain an *outlist* for each partition. An outlist is a list of inter-partition references contained in the partition.

The following invariants guarantee that only unreachable objects are collected:

1. All external references contained in a partition are included in the outlist.
2. The count of a reference in an inlist is equal to the number of outlists containing the reference.

Inlists and outlists are stored on disk so that they can be recovered quickly after a failure. They can be maintained as regular heap objects, separate from the partition's segments.

When an inter-partition reference is created from partition p to q due to a modification, the outlist for p and the inlist for q need to be updated to preserve the invariants. Our scheme defers reading or writing the disk to record these updates by using small, *potential* inlists and outlists in memory.

3.2 Potential Inlists and Outlists

Modified objects in the log are scanned lazily to find inter-partition references. When an object in partition p is scanned, we record any external references in the *potential outlist* for p, and we also update the *potential inlists* of the target partitions. Each entry in a potential inlist contains a reference count that counts the number of potential outlists that contain the reference. To distinguish inlists and outlists from their potential counterparts, we refer to the former as the *basic* lists. The following revised invariants still guarantee safety:

1. All external references contained in a partition are included in the basic or potential outlist (or both).

2. (a) The count of a reference in a basic inlist is equal to the number of basic outlists containing the reference.
   (b) The count of a reference in a potential inlist is equal to the number of potential outlists containing the reference.

Invariant 1 is maintained lazily; it holds when the log is fully processed. The constraints are that before collecting a partition, the log must be processed to generate the full potential inlist of the partition, and that modified objects must be processed before they are truncated from the volatile log.

The potential lists grow slowly because there are expected to be relatively few inter-partition references in modified objects. Further, references already present in the old values need not be added to the potential lists. For example, in transactional systems, old copies of modified objects are retained in case the transaction aborts; this information can be used to avoid unnecessary additions to potential lists. However, there may still be overlap between the potential and basic outlists. When potential lists grow too big, we merge them into the basic lists.

3.3 Merging Potential and Basic Lists

We move entries from potential lists to basic lists in batches. This is a two step process: we merge the outlists first and merge the inlists later.

When the total size of the potential outlists grows beyond a certain limit, we select a few partitions with the largest potential outlists, and read in their basic outlists. References in a potential outlist that are not already present in the basic outlist are added to the basic outlist. The potential outlist is then discarded and corresponding potential inlist counts are decremented to maintain Invariant 2b.

Updating a single basic outlist can require increments to entries in several different basic inlists. Reading in these inlists at this point would result in several disk accesses. Therefore, we record the increments to the basic inlists in yet another data structure in memory called the *delta* inlist.

A delta inlist contains a set of references with associated counts. Unlike potential inlists, which contain potential increments, delta inlists contain *definite* increments to the basic inlists. Invariant 2a is then revised to the following:

> (2a) The count of a reference in a basic inlist plus that in the delta inlist is equal to the number of basic outlists containing the reference.

When the total size of delta inlists grows beyond a certain limit, we merge some of them into the basic inlists. We select a few partitions with a largest delta inlists, read in their basic inlists, add the counts in the delta inlists to the basic inlists, and discard the delta inlists. The generation of the various lists is shown in Figure 1.

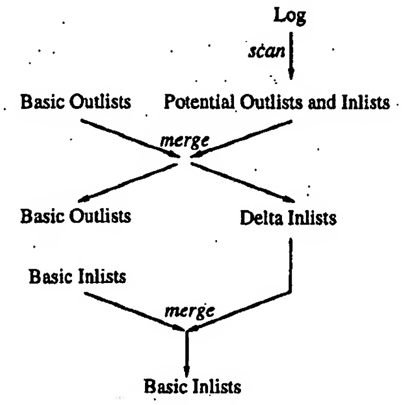

Figure 1: Data structures to batch disk accesses

3.4 Collecting a Partition

Any policy may be used to select partitions for collection. (Studies have shown that it is desirable to be flexible in selecting partitions [6].) To trace a selected partition, we load its segments into the cache and process the log completely to generate the partition's full potential inlist. Since a partition is a fraction (say, 10%) of the cache, it does not disturb the cache much when loaded.

The Roots. We include the following in the root set of a partition:

1. The persistent root of the heap.
2. Roots from applications, such as variables.
3. The basic inlist for the partition, which is read in from the disk. If there is a delta inlist, it is merged with the basic inlist and discarded.
4. The potential inlist for the partition, if any.

Obtaining application roots depends on the specific system model; special care is needed in systems where applications have caches of their own, but we ignore this issue in this paper and assume that application roots are readily available.

Compaction. If the collector compacted storage by moving objects among segments it would be necessary to fix up references to moved objects that are stored in other partitions as well as in application variables. Most generational schemes solve this problem by remembering the exact objects or locations in other partitions that refer to a given object [20], but this is too much information to maintain in a large heap. Therefore, we compact objects within each segment and thus preserve object names, as in [1]. Note that objects that are not referenced from other partitions or the applications (as indicated by the root set) could indeed be moved to other segments within the partition.

Our scheme could also be used in systems that do not have opaque names within segments (e.g., systems that store virtual memory pointers in objects), but in that case it would not be possible to do compaction.

Tracing Scheme. Our approach can be used in combination with various tracing schemes. For example, we could use a replicating collector like that described in [16], in which applications access the old copies of segments while the collector is generating the new ones with the aid of the modification log. Such a scheme requires little synchronization with applications, but needs space for two partitions in primary memory.

A mark-and-sweep scheme that rescans modified objects in the log can be used as well. Such an approach requires less space than copying. No synchronization is needed during the mark phase. The sweep phase can compact one segment at a time, either by locking the segment from applications and sliding objects or by making a new copy of the segment to replace the old one. No work is needed for segments with no garbage objects; this may be a significant advantage over copying collection in persistent stores where little garbage is created.

Updating the Outlist. As a partition is traced, the collector generates a new basic outlist for it. After collection, it compares the old and the new outlists, increments the corresponding delta inlist entry for every new reference, and decrements the delta inlist entry for every missing reference. Thus there may be negative counts in delta inlists. These steps ensure that Invariant 2a is maintained.

The new basic outlist replaces the old outlist. The potential outlist, if present, is discarded and the corresponding counts in the potential inlists are decremented.

3.5 Crash Recovery

It would be lots of work to recompute inlists and outlists after a crash. Therefore, we store basic inlists and outlists as persistent objects. Their modifications are logged (e.g., at the end of collection), and are installed on disk later as with normal objects.

Information in potential and delta lists must also be recoverable after a crash to preserve the invariants. When the stable log is truncated, potential outlist information for the truncated records is first written to the log to preserve Invariant 1. Potential outlist information for records that have not been truncated is not logged; instead this information is recovered by reprocessing the log after a crash. When a potential outlist is merged with a basic outlist, any log records for the potential outlist are deleted (by writing a deletion record after logging the new value of the basic outlist). Potential inlists are never logged; they are recomputed from potential outlists on recovery, thus preserving Invariant 2b.

When a delta inlist is updated (as a result of updating a basic outlist), the updated values of the delta inlist and the basic outlist are logged atomically to preserve Invariant 2a. When a delta inlist is merged with a basic inlist, the log records for the delta inlist are deleted (by writing a deletion record after logging the new value of the basic inlist).

The segments of a garbage-collected partition can be independently flushed to the disk. Thus all the collection state that was available before a failure can be quickly recovered afterwards and the log is used to reduce disk I/O's associated with storing the needed information.

3.6 Safety and Liveness

Invariants 1, 2a, and 2b guarantee that objects reachable from other partitions will not be collected: If there is a reference r from partition p to q, then Invariant 1 implies that r must be in the basic or potential outlist for p. If r is in the basic outlist for p, Invariant 2a implies that the sum of the counts for r in the delta and basic inlists for q is at least one. If r is in the potential outlist for p, Invariant 2b implies that the count for r in the potential inlist for q is at least one. In either case, r will be included in the root set for q.

Further, our scheme is guaranteed to collect all garbage eventually—except for inter-partition cyclic garbage, which is collected using a separate scheme. An unnecessary entry in a basic or potential outlist will be removed when its partition is next collected and the corresponding inlist count will be decremented. This guarantees that objects not reachable from the roots are collected. From this it can be shown inductively that if the partitions are collected periodically, all garbage except for inter-partition cyclic garbage will be collected.

4 Related Work

All partitioned collection schemes, whether designed for centralized or distributed systems, require techniques to maintain inter-partition reference information efficiently. In particular, maintaining this information for a large heap on disk requires efficient use of memory and disk. However, the only other work that addresses this issue is PMOS by Moss *et al.* [7, 15].

PMOS collects segments independently; a segment is the unit of fetching and tracing. Each segment contains an inlist, which identifies the source segments that contain references to any given object; such inlists take more space than reference counts. On the other hand, outlists are not stored on disk; instead, whenever a segment is read into the cache, it is scanned to compute its outlist. When a modified segment is evicted, it is scanned again to compute differences from the old outlist; the differences are stored in an object equivalent to our delta inlists to avoid disk accesses. Our use of basic outlists on disk, the log, and the potential lists avoids processing segments when they are fetched or evicted.

PMOS compacts objects across segments. This provides better compaction, but it necessarily changes the names of moved objects. When an object moves, cached segments that contain references to it are scanned and updated (the inlist identifies such segments). Segments fetched later have to be similarly updated. Further, when an object moves, inlist entries for every inter-segment reference it contains must be updated to identify the new source segment. We avoid these costs by compacting objects within their segments. Furthermore, while PMOS must trace objects using a copying scheme, we can use either mark-and-sweep or copying.

PMOS collects inter-segment cyclic garbage by grouping segments into *trains*, and gradually migrating all reachable objects in a train to other trains such that the train contains only cyclic garbage at the end and can be discarded. (This approach relies on being able to fix references to moved objects.) While copy-collecting a segment in a train, objects are moved to the newest segments in the trains that refer to them. Thus, multiple segments may be accessed while collecting a segment.

Our scheme collects inter-partition cyclic garbage using an incremental global marking scheme that preserves the locality of partitioned collection [14]. While global marking is not fault tolerant in a distributed system and requires the cooperation of all machines, these problems do not arise on a single machine and in that environment tracing is likely to be more efficient than moving objects for collecting inter-partition garbage cycles.

5 Conclusion

This paper has described a garbage collection scheme for large persistent object stores that makes efficient use of the disk and main memory and does not delay the application. The heap is divided into partitions that are collected independently using inlists and outlists that are kept on disk to save space in memory and to make crash recovery fast. We use novel techniques to maintain this information using compact in-memory data structures and a log to avoid disk accesses. The scheme has been extended with a marking technique that allows us to also collect inter-partition cyclic garbage without delaying the collection of any other garbage. Thus the scheme collects all garbage yet preserves the localized and scalable nature of partitioned collection.

Acknowledgements

We are grateful to Atul Adya and Miguel Castro for proofreading this paper, and to the referees for their comments.

References

[1] L. Amsaleg, O. Gruber, and M. Franklin. Efficient incremental garbage collection for workstation–server database systems. In *Proc. 21st VLDB*. ACM Press, 1995.

[2] H. G. Baker. List processing in real-time on a serial computer. *CACM*, 21(4):280–94, 1978.

[3] H. G. Baker. 'Infant mortality' and generational garbage collection. *ACM SIGPLAN Notices*, 28(4), 1993.

[4] P. B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report MIT/LCS/TR–178, MIT, 1977.

[5] J. E. Cook, A. W. Klauser, A. L. Wolf, and B. G. Zorn. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proc. 1996 SIGMOD*. ACM Press, 1996.

[6] J. E. Cook, A. L. Wolf, and B. G. Zorn. Partition selection policies in object databases garbage collection. In *Proc. 1994 SIGMOD*. ACM Press, 1994.

[7] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[8] E. K. Kolodner and W. E. Weihl. Atomic incremental garbage collection and recovery for large stable heap. In *Proc. 1993 SIGMOD*, pages 177–186, 1993.

[9] R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *Proc. International Conference on Distributed Computing Systems*. IEEE Press, 1992.

[10] B. Lang, C. Queinniec, and J. Piquer. Garbage collecting the world. In *Proc. POPL '92*, pages 39–50. ACM Press, 1992.

[11] H. Lieberman and C. E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 26(6):419–29, 1983.

[12] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. 1996 SIGMOD*. ACM Press, 1996.

[13] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In *Proc. 3rd Parallel and Distributed Information Systems*. IEEE Press, 1994.

[14] U. Maheshwari, B. Liskov, and T. Ng. Partitioned garbage collection of a large stable heap. Technical Report MIT/LCS/TR–699, MIT LCS, 1996.

[15] J. E. B. Moss, D. S. Munro, and R. L. Hudson. Pmos: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proc. 7th Workshop on Persistent Object Systems*, 1996.

[16] S. M. Nettles, J. W. O'Toole, D. Pierce, and N. Haines. Replication-based incremental copying collection. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[17] T. Ng. Efficient garbage collection for large object-oriented databases. Technical Report MIT/LCS/TR–692, MIT LCS, 1996.

[18] J. W. O'Toole, S. M. Nettles, and D. Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proc. 14th SOSP*, pages 161–174, 1993.

[19] M. Shapiro and P. Ferreira. Larchant–RDOSS: a distributed shared persistent memory and its garbage collector. In *Proc. Workshop on Distributed Algorithms*, number 972 in Lecture Notes in Computer Science, pages 198–214. Springer-Verlag, 1995.

[20] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[21] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in clinet–server persistent object stores. In *Proc. Data Engineering*, pages 120–133. IEEE Press, 1994.